C++ Move Semantics for Exception Safety and Optimization in Software Transactional Memory Libraries

Justin E. Gottschlich, Jeremy G. Siek, and Daniel A. Connors

University of Colorado at Boulder, Boulder, CO, USA
{gottschl, jeremy.siek, dconnors}@colorado.edu

Abstract. Software transactional memory (STM) is an effective near-term solution for gaining experience with transactional memory (TM) and relies on either traditional lock-based synchronization or non-blocking techniques. While non-blocking techniques naturally scale, lock-based approaches require fewer changes to user-defined data structures. Some lock-based STM libraries use user-defined or compiler generated copy assignment operators to reduce intrusiveness on user-defined classes. Unfortunately, such operations may throw exceptions and prevent the commit or abort of a transaction from completing, thereby leaving the program in a partially committed or inconsistent state. We present a solution to this problem: our STM library *moves* objects back to global memory instead of copying them. The ability to move objects is a new feature in the next version of the C++ standard, supported by the addition of *rvalue references*. We demonstrate our solution within DracoSTM, a C++ lock-based STM library, and present a performance comparison of move versus copy for both direct and deferred updating.

1 Introduction

Transactional memory (TM) reduces the complexity of parallel programming [8, 13] and delivers performance that is competitive with other synchronization mechanisms [3, 4]. With hardware trends leading to more CPUs, not faster CPUs, there is need for parallel programming methods that are less error-prone and easier to master [5, 15]. A *transaction* is a programmer-specified task that encapsulates a number of operations, making them appear as one atomic action.

Deferred versus Direct Update. There are two different policies for catching and repairing transactional conflicts: *deferred updated* and *direct update*. In the deferred update approach, the transaction makes a local copy of global memory and works on local copy. On writes, the transaction updates the local copy. On commits, the transaction copies the local memory to global memory and on aborts, the transaction discards the local copy. In the direct update approach, the transaction makes a local backup of the global memory. On writes, the transaction directly updates global memory. On commits, the transaction discards the backup and on aborts, the transaction copies the backup to global memory.

Lock-based Designs and Exceptions in C++. Lock-based C++ STM libraries, such as DracoSTM, may rely on the copy assignment operator of user-defined classes to commit a transaction's local data to global memory (in the deferred update policy), or to rollback the global state on transactional aborts (in the direct update policy). Unfortunately, copy assignment operators may need to allocate memory or perform some other exception emitting behavior resulting in thrown exceptions [14]. If such an exception occurs while a transaction is committing, the result is a partially committed transaction. Even more troubling is the direct update case, where global memory is left in a terminal state if an exception is thrown during the rollback phase. Consider the following example use of DracoSTM.

```
1  struct people : public transaction_object<people>
2  { vector<string> first, last; };
3  people gPeople;
4
5  void insert_all(vector<string>& fs, vector<string>& ls) {
6    for (transaction t;;)
7      try {
8         for (int i = 0; i != fs.size(); ++i) {
9           t.write(gPeople).first.push_back(fs[i]);
10          t.write(gPeople).last.push_back(ls[i]);
11        }
12        t.end_transaction(); return;
13     } catch (aborted_transaction&) {}
14 }
```

Upon the first `t.write()` of the transaction, DracoSTM makes a full copy of `gPeople`. During transaction execution, the original `gPeople` is modified. Now suppose that the transaction aborts because there is a conflict with another transaction. If there is not enough memory available to copy the original `gPeople` back, an exception will be thrown. The global `gPeople` will remain in an inconsistent state, reflecting changes made by the aborted transaction.

We solve the problem of commit-time and abort-time exceptions by using move semantics in place of copy semantics. The idea of moving is new to C++ and will be available in the next version of the standard. Move semantics are implemented, in part, by using a new language extension called rvalue references.

Technical Contributions. This work includes the following original technical contributions:

- The main contribution of this work is the use of move semantics to overcome commit-time and abort-time exceptions in lock-based STM systems.
- We present experimental results detailing the performance differences between copy and move semantics for both direct and deferred updating.

2 Background

When DracoSTM commits a deferred updating transaction or aborts a direct updating transaction it performs an object copy via `copy_state()` which invokes the copy assignment operator for the class derived from `transaction_object`. The calls to the copy assignment operator are needed for restoring global memory when a direct updating transaction aborts or updating global memory when a deferred updating transactions commits. In either case, the call to the copy assignment operator may throw an exception. These thrown exceptions can strike at untimely points in a transaction's lifetime.

Exceptions occurring while deferred updating transactions commit are unacceptable. Such exceptions would result in partially committed transactions. While this behavior is similar to C++'s STL behavior [1, 12] for identical reasons (copy constructor failures), such exceptions are unacceptable in transactional memory as the exception would break the transaction's atomicity.

Likewise, exceptions occurring when direct updating transactions abort are also unacceptable. Direct update abort-related exceptions are terminal; failing to restore the global state of the program when a transaction has been aborted violates program correctness. Since the transaction did not commit, any lingering transactional state in global memory is inconsistent and the program must immediately halt.

3 Move Semantics in C++

In this section, we describe move semantics and the motivation behind their integration in the next version of C++. C++ move semantics is a mechanism that enables objects to take ownership of other objects in an optimized manner that has implicit exception safety [9]. Given two objects, x and y, moving enables the transfer of the contents of x to y, or vice versa, in an optimized manner that is allowed to alter the state of both x and y [9]. Consider the following:

```
1  void fun(vector<int> &retVect) {
2    vector<int> locVect;
3    for (int i = 0; i < max; ++i) locVect.push_back(calc(i));
4    retVect = locVect;
5  }
```

Above, the `vector<int> locVect` is constructed within `fun()` and is populated with values returned from `calc()`. Once `max` values are placed in `locVect`, the contents of `locVect` are copied into `retVect`. The goal of the above code is to have `retVect` represent the same vector as `locVect` and while performing a copy of `locVect` is one way to achieve this it has unnecessary copy overhead which degrades performance. One solution to reduce the copy overhead in `fun()` is to use `retVect` in place of `locVect`:

```
1  void fun(vector<int> &retVect) {
2    for (int i = 0; i < max; ++i) retVect.push_back(calc(i));
3  }
```

While directly using `retVect` removes copy overhead, it introduces exception safety concerns. If an exception is thrown during the `for` loop, `retVect` will be placed in an unknown state. Therefore, replacing `locVect` with `retVect` is not ideal. Another solution is to replace the copy assignment of `retVect` with a `swap()`:

```
1  void fun(vector<int> &retVect) {
2    vector<int> locVect;
3    for (int i = 0; i < max; ++i) locVect.push_back(calc(i));
4    swap(retVect, locVect);
5  }
```

Yet, `swap()` incurs the construction overhead of a temporary object unless it is specialized. While `vector<>` containers are among those containers specialized, types which are not specialized incur temporary construction overhead. Specializing `swap()` for all possible types is not feasible [11].

A solution to the above problem should probably include (1) direct language-support, (2) removal of copy overhead and (3) high exception safety. The next version of C++ introduces two new operations to fulfill such behavior: move construction and move assignment. Move constructors and move assignments overload the pre-existing copy constructor and copy assignment operators with a signature change using an rvalue reference parameter, denoted by `&&`, instead of an lvalue reference parameter, denoted by `&` [2]. The below example shows a class using both copy and move semantics.

```
1  class A {
2     int *data;
3  public:
4    A(int v) { data = new int; *data = v; }
5    ~A() { delete data; }
6
7    // copy semantics
8    A(A &a) { data = new int; *data = a.val(); }
9    A& operator=(A& a) { *data = a.val(); return *this; }
10
11   // move semantics
12   A(A &&a) : data(a.data) { a.data = 0; }
13   A& operator=(A&& a) { swap(data, a.data); return *this; }
14
15   int val() { return data == 0 ? 0 : *data; }
16 };
```

The lvalue reference operations, lines 7-9, perform copy semantics and ensure the right-hand side (rhs) object's state is unchanged [9], while the rvalue reference operations, lines 11-13, perform move semantics and intentionally change the rhs object's state either by "zeroing it out" as done in the move constructor or by swapping its state with the left-hand side (lhs) as is done in the move assignment. Due to the implementation, the copy semantics of the above class

can throw exceptions, but the move semantics cannot. No `new` operations (which can throw `bad_alloc` exceptions) are required in the move constructor or assignment nor can the remaining operations, which are simple integer sets or swaps, throw exceptions. The nothrow behavior of move assignment is a fundamental requirement for DracoSTM to update global memory while ensuring exception-less behavior. Moreover, not only do move semantics provide an exception safe way to overcome the direct update abort and deferred update commit problem in DracoSTM, they also more correctly supply what is necessary for transactional memory updates, improving overall system performance by removing unnecessary copy behavior.

4 DracoSTM & C++ Move Semantics

In this section we describe how DracoSTM's system was expanded to support C++ move semantics. DracoSTM's move semantic integration can be broken into two categories: (1) library integration and (2) user-defined transaction integration. Library integration of move semantics requires a one-time change to DracoSTM, while user-defined transaction integration is a continual process.

4.1 Library Integration

The DracoSTM library integration of move semantics required the addition of a `move_state()` method to the `transaction_object<>`. The `move_state()` method is internally called and invokes the user-defined derived transaction class's move assignment (e.g. `operator=(type &&)`). The details of the `move_state()` method as implemented in DracoSTM are shown below. In addition, we also show our implementation of the `move()` function template, as only rvalue references are currently available in ConceptGCC, not `move()` operations [7]. Our `move()` implementation follows the specification described by Hinnant et al. in [11].

```
1  namespace std {
2    template <class T> typename std::remove_reference<T>::type&&
3    move(T&& t) { return t; }
4  }
5
6  // support for move_state()
7  template <class Derived>
8  class transaction_object : public base_transaction_object {
9  public:
10   virtual void move_state(base_transaction_object * rhs)
11   {
12     static_cast<Derived &>(*this) = std::move
13       (*(static_cast<Derived*>(rhs)));
14   }
15 };
```

Four `transaction` interfaces were added for client control of move semantics:

- `void enable_move_semantics()` - turns on move semantics and turns off copy semantics.
- `void enable_copy_semantics()` - turns on copy semantics and turns off move semantics.
- `bool using_move_semantics()` - returns true if DracoSTM is using move semantics for transactions, otherwise it returns false.
- `bool using_copy_semantics()` - returns true if DracoSTM is using copy semantics for transactions, otherwise it returns false.

Two additional changes were required in the deferred updating commit and direct updating abort methods, but have been omitted for brevity.

4.2 User-Defined Transaction Integration

Move constructors and move assignments will not be synthesized by the compiler [6, 10] requiring users to define their own move constructors and assignments. However, Glassborow and Goldthwaite have presented a proposal to automate move constructors and assignments through explicit user requests [6]. Until move semantic automation becomes available, users will be required to define their own move constructors and assignments. The below example demonstrates a DracoSTM client-defined transaction class implementing move assignment as necessary for DracoSTM.

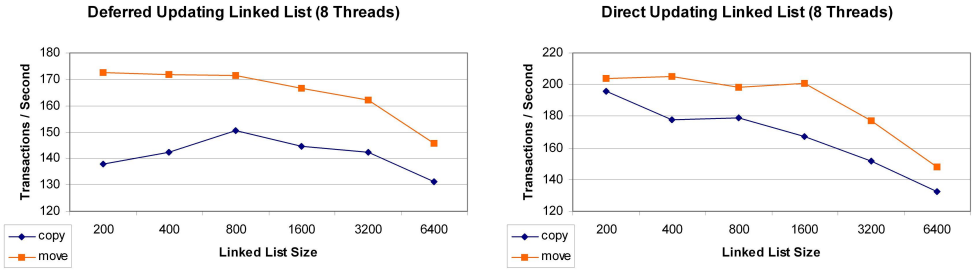
```
1  template <class T> class list_node :
2    public transaction_object< list_node<T> >
3  {
4  public:
5    // move assignment
6    list_node& operator=(list_node&& rhs) {
7      value_ = std::move(rhs.value_);
8      std::swap(next_, rhs.next_);
9      return *this;
10   }
11 private:
12   list_node *next_;
13   T value_;
14 };
```

If Glassborow and Goldthwaite's proposal [6] to support automatic generation of move semantics through explicit user request is accepted into the C++ standard the overhead of implementing move assignment will be trivial. As such, using move assignment to overcome exceptions in direct and deferred updating may result in an ideal manner to overcome the open exception problem with lock-based STM designs using copy assignment to update global memory.

5 Experimental Evaluation

Our experimental evaluation explored 8 threaded linked lists to evaluate the performance difference between copy and move semantics. All evaluations were

performed on a 3.2 GHz 4-processor Intel Xeon with 16GB of RAM using the ConceptGCC compiler (version 4.3.0 alpha 6). The linked list benchmarks contains nodes of transaction classes, each of which contain their own vector of 10,000 or 100,000 elements. The tests using nodes with 10,000 vector elements per node show less than 12% performance difference between copy and move semantics for all tests. The benchmarks using nodes containing 100,000 elements per node display more noticeable performance differences and are shown in Figure 1(a) and 1(b). Figure 1(a) shows linked list performance when using deferred updating. Figure 1(b) shows linked list performance when using direct updating.



(a) Deferred update linked list

(b) Direct update linked list

Fig. 1. Linked list benchmarks.

Deferred updating transactions displayed noticeable performance differences when (1) move semantics were used and (2) copy construction or assignment overhead was high. For the 100,000 element nodes, the performance results for deferred updating showed move semantics outperform copy semantics by a margin of 25%, 21%, 14%, 15%, 14% and 11% for doubling sized linked lists starting at 200 initial nodes.

Direct updating performance differences between copy and move semantics were expected to be minor because direct updating can only take advantage of the move semantic performance benefits when transactions abort. As transactional aborts are much more infrequent than transactional commits (on average 95% of all transactions commit), intuition suggests the performance gain would be minimal. However, direct updating (as shown in Figure 1(b)) yielded 5%, 15%, 11%, 20%, 17% and 11% speed improvement for move semantics over copy semantics. These results seem considerable given the relatively low abort rate of the linked list at $\approx 5\%$.

6 Conclusion and Future Work

We presented a novel solution using move semantics to overcome the side-effects of exceptions during deferred updating commits and direct updating aborts.

Move semantics overcome partially committed transactions and program terminating behavior by eliminating exceptions during deferred updating commit-time and direct updating abort-time, respectively. The exceptions at commit-time and abort-time are eliminated by using only *non-throwable* operations with move assignment.

References

1. D. Abrahams. Exception-safety in generic components. *Online: www.boost.org /more /generic_exception_safety.html*.
2. D. Abrahams, P. Dimov, D. Gregor, H. E. Hinnant, A. Hommel, and A. Meredith. Impact of the rvalue reference on the standard library. Technical Report N1771=05-0031, Mar 2005.
3. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In S. Dolev, editor, *Distributed Computing (20th DISC'06)*, volume 4167 of *Lecture Notes in Computer Science (LNCS)*, pages 194–208. Springer-Verlag (New York), Stockholm, Sept. 2006.
4. D. Dice and N. Shavit. What really makes transactions faster? *ACM SIGPLAN Workshop on Transactional Computing*, June 2006.
5. K. Fraser and T. Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst*, 25(2), 2007.
6. F. Glassborow and L. Goldthwaite. Explicit class and default definitions. Technical Report N1717 04-0157, 2004.
7. D. Gregor. Decltype and rvalue references and variadic templates, oh my! Technical report, Feb 2007.
8. T. Harris, S. Marlow, S. L. P. Jones, and M. Herlihy. Composable memory transactions. In K. Pingali, K. A. Yelick, and A. S. Grimshaw, editors, *PPOPP*, pages 48–60. ACM, 2005.
9. H. E. Hinnant. A proposal to add move semantics support to the C++ language. Technical Report N1377=02-0035, Sep 2002.
10. H. E. Hinnant. Some design rationale for the rvalue references. Technical report, May 2006.
11. H. E. Hinnant, B. Stroustrup, and B. Kozicki. A brief introduction to rvalue references. Technical Report N2027=06-0097, Jun 2006.
12. N. M. Josuttis. *The C++ Standard Library: a tutorial and reference*. Addison-Wesley, pub-AW:adr, 1999.
13. N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the 14th ACM Symposium on Principles of Distributed Computing*, pages 204–213. Aug 1995.
14. B. Stroustrup. *The C++ Programming Language (3rd Edition)*. Addison & Wesley, Reading, Mass., 1997.
15. H. Sutter and J. Larus. Software and the concurrency revolution. *ACM Queue*, September 2005.